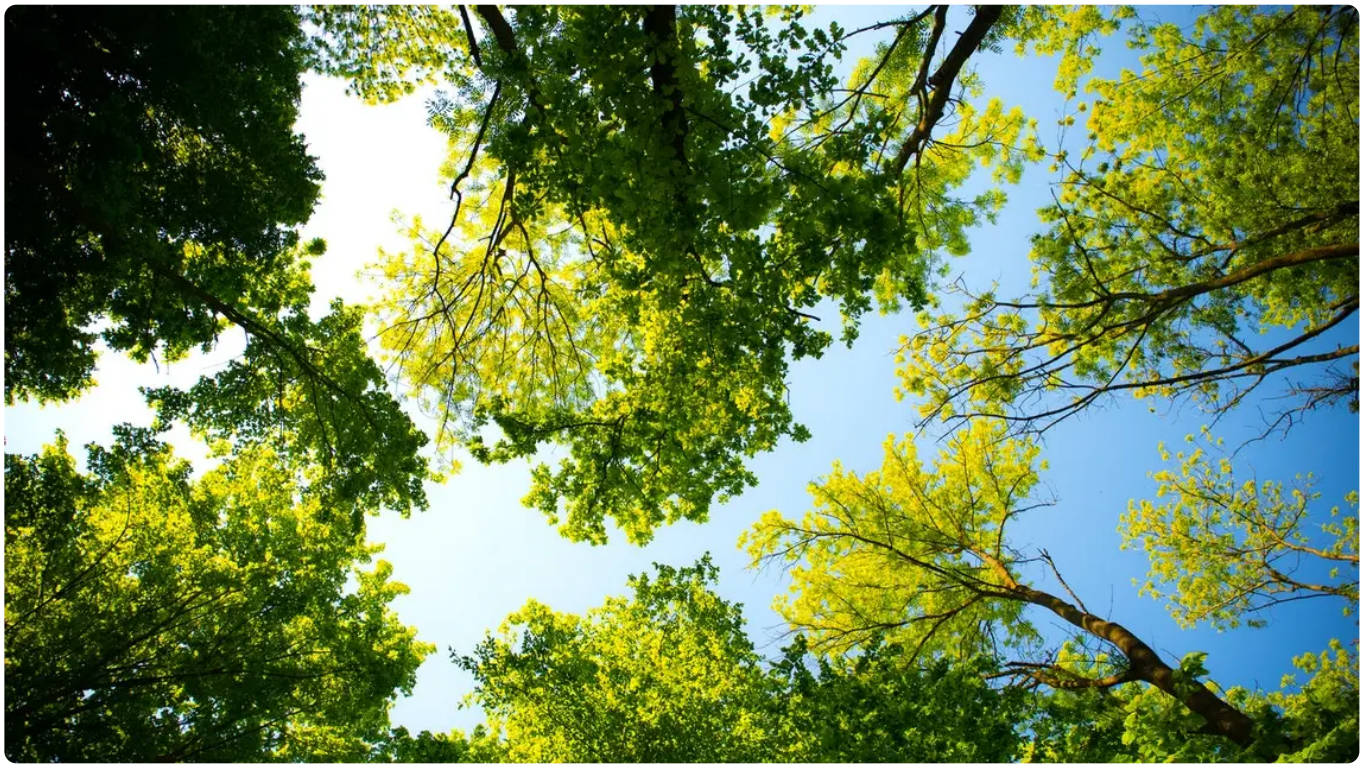
下载APP

07｜整合数据：枚举、结构与联合是如何实现的？

2021-12-22 于航

《深入C语言和程序运行原理》 课程介绍 >



讲述：于航

时长 12:06 大小 11.09M

你好，我是于航。

C 语言为我们提供了高于机器指令的一定抽象能力，这使得我们能够以接近自然语言的方式来构建应用程序。如果说使用 C 语言是用砖块来造房子，那使用其他高抽象粒度编程语言，就是直接以墙面为单位来搭建。很明显，从这个角度来说，C 语言用起来不如其他高级语言方便，但它也同时给予了更细的构建粒度，让我们能够按照自己的想法，灵活自义墙面的形态。

领资料

对于这里提到的砖块和墙面，你可以将它们简单理解为编程语言在构建程序时使用的数类型。比如在 Python 语言中，我们可以使用集合（set）、字典（dict）等复杂数据类型。而在 Java 语言中，Map 本身又会被细分为 HashMap、LinkedHashMap、EnumMap 等多种类型，供不同应用场景使用。

为了在保持自身精简的同时也保证足够高的灵活性，C 语言在提供基本数值类型和指针类型的基础上，又为我们提供了结构（struct）、联合（union）与枚举（enum）这三种类型。结合使用这些类型，我们就能将小的“砖块”组合起来，从而将它们拼接成为更大的、具有特定功能结构的复杂构建单元。

接下来，就让我们一起看看：编译器是如何在背后实现这三种数据类型的？而在实现上，为了兼顾程序的性能要求，编译器又做了哪些特殊优化？

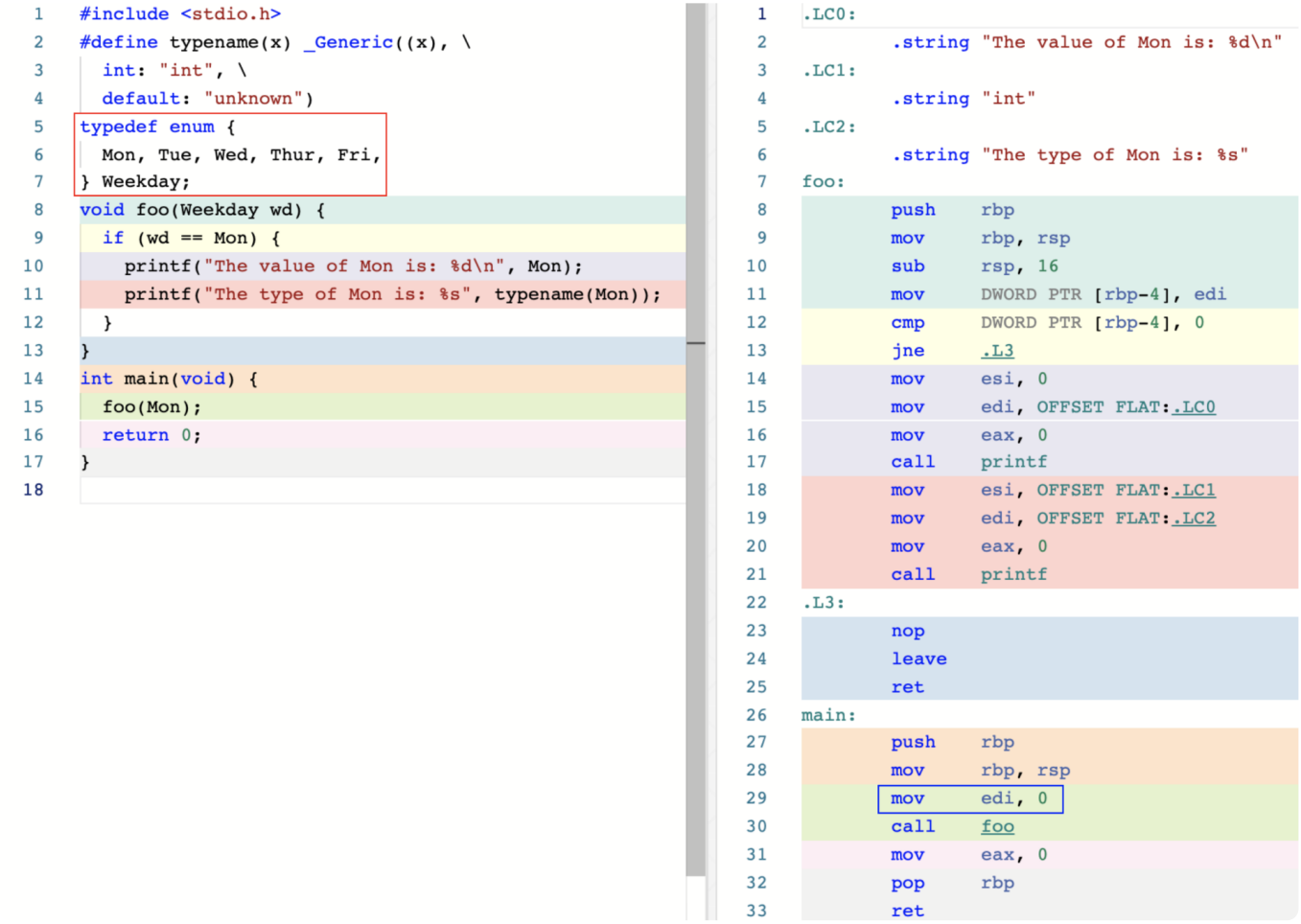
枚举

在编程语言中，枚举（Enumeration）这种数据类型可以由程序员自行定义，用来表示某类可取值范围有限的抽象概念。

下面我们来看一个经典的例子：应该如何使用编程语言来表示 “周工作日（weekday）”这个概念呢？

周工作日属于现实世界中的一种抽象概念，它包含周一到周五共五个有效值。不同于数值、字符等概念，它无法直接对应到物理计算机中的任何软硬件实现上。因此，为了能够在程序中更加精确地表达这类信息，我们可以用枚举来自定义对应的类型。

在 C 语言中，我们可以这样实现：



```c
#include <stdio.h>
#define typename(x) _Generic((x), \
  int: "int", \
  default: "unknown")
typedef enum {
  Mon, Tue, Wed, Thur, Fri,
} Weekday;
void foo(Weekday wd) {
  if (wd == Mon) {
    printf("The value of Mon is: %d\n", Mon);
    printf("The type of Mon is: %s", typename(Mon));
  }
}
int main(void) {
  foo(Mon);
  return 0;
}
```

```asm
.LC0:
        .string "The value of Mon is: %d\n"
.LC1:
        .string "int"
.LC2:
        .string "The type of Mon is: %s"
foo:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     DWORD PTR [rbp-4], edi
        cmp     DWORD PTR [rbp-4], 0
        jne     .L3
        mov     esi, 0
        mov     edi, OFFSET FLAT:.LC0
        mov     eax, 0
        call    printf
        mov     esi, OFFSET FLAT:.LC1
        mov     edi, OFFSET FLAT:.LC2
        mov     eax, 0
        call    printf
.L3:
        nop
        leave
        ret
main:
        push    rbp
        mov     rbp, rsp
        mov     edi, 0
        call    foo
        mov     eax, 0
        pop     rbp
        ret
```

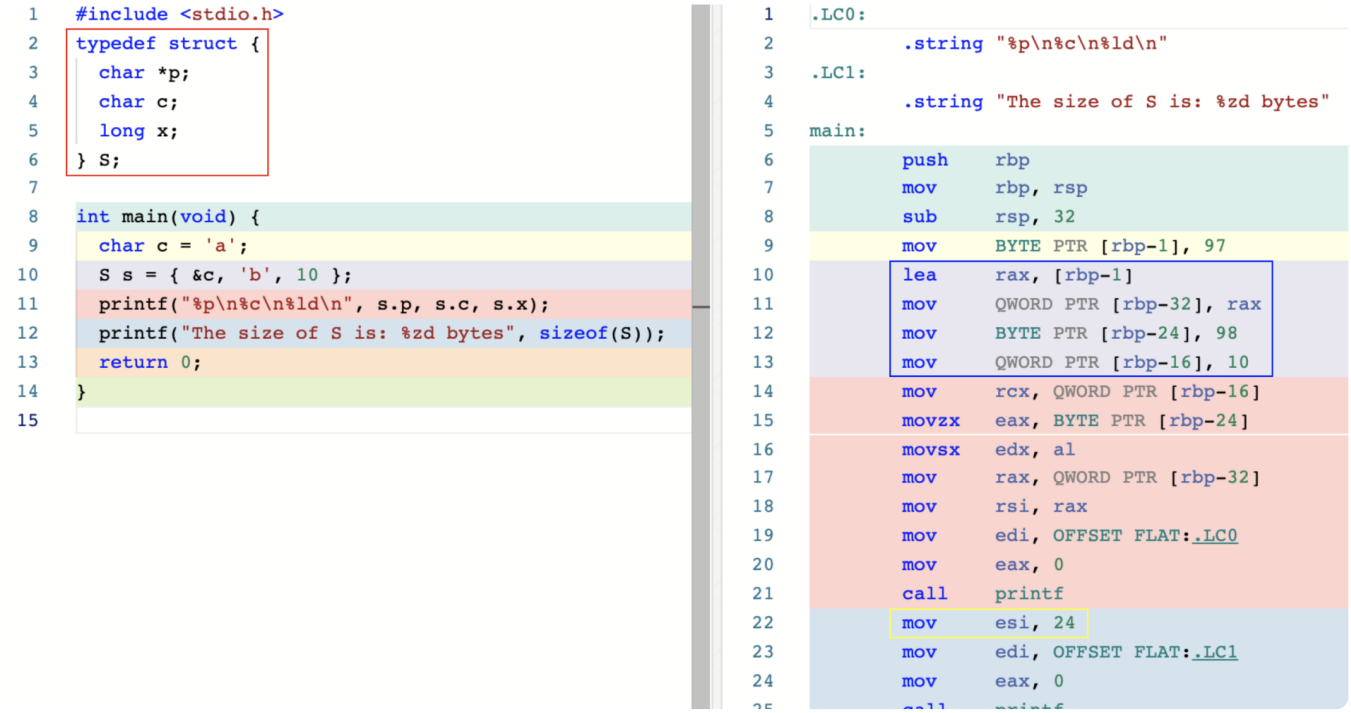
为了便于观察，我直接展示了 C 代码及其对应的汇编代码。可以看到，编译器没有为左侧红框内的枚举类型定义生成任何的机器指令。实际上，在 C 语言中，每一个自定义枚举类型中的枚举值，都是以 `int` 类型的方式被存储的，因此，这些枚举值有时也被称为“具名整型”。你可以从上图右侧蓝框内的汇编代码中看到，当函数 foo 被调用时，传入的枚举值 Mon 正对应于通过 edi 寄存器传入的字面量数字 0。也就是说，枚举值 Mon 在底层是由数字值 0 表示的。

同样地，在左侧 C 代码的第 11 行，我们也使用了泛型宏来判断枚举值 Mon 的具体类型。你可以尝试运行这段代码，并观察程序的输出结果，以验证我们的结论。

需要注意的是，**C 标准直接将枚举值当作整数进行处理的这种方式，可能会导致我们在构建程序时遇到意想不到的问题。**比如，对于上述这段 C 代码，函数 foo 在被调用时，实际上允许传入任何可以被隐式转换为 `int` 类型的值，哪怕这个值来源于另一个枚举类型的变量。因此，让枚举类型有助于组织程序代码的同时并确保它不被乱用，也是我们在构建高质量程序时需要注意的一个问题。

结构

在 C 语言中，数组用来将一簇相同类型的数据存放在连续的内存段上。而结构（Struct）实际上与其类似，只不过在结构内部，我们可以存放不同类型的数据。先来看一段代码：



```c
#include <stdio.h>
typedef struct {
  char *p;
  char c;
  long x;
} S;

int main(void) {
  char c = 'a';
  S s = { &c, 'b', 10 };
  printf("%p\n%c\n%ld\n", s.p, s.c, s.x);
  printf("The size of S is: %zd bytes", sizeof(S));
  return 0;
}
```

```
.LC0:
        .string "%p\n%c\n%ld\n"
.LC1:
        .string "The size of S is: %zd bytes"
main:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 32
        mov     BYTE PTR [rbp-1], 97
        lea     rax, [rbp-1]
        mov     QWORD PTR [rbp-32], rax
        mov     BYTE PTR [rbp-24], 98
        mov     QWORD PTR [rbp-16], 10
        mov     rcx, QWORD PTR [rbp-16]
        movzx   eax, BYTE PTR [rbp-24]
        movsx   edx, al
        mov     rax, QWORD PTR [rbp-32]
        mov     rsi, rax
        mov     edi, OFFSET FLAT:.LC0
        mov     eax, 0
        call    printf
        mov     esi, 24
        mov     edi, OFFSET FLAT:.LC1
        mov     eax, 0
```

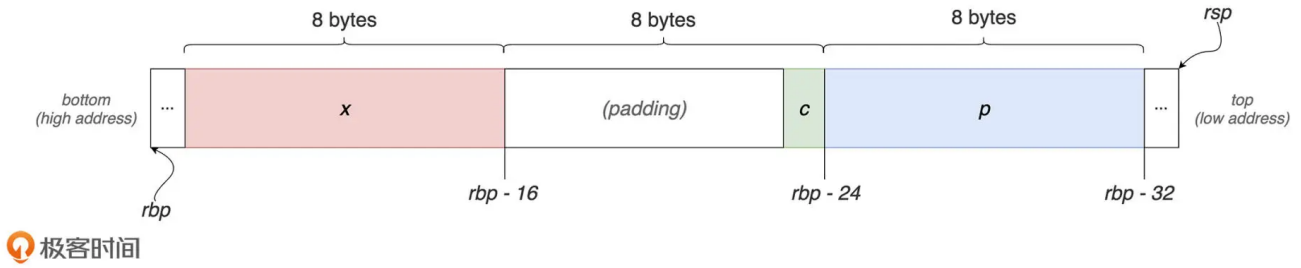
在上图左侧的 C 代码中，我们定义了一个名为 S 的结构。对于每一个结构 S 的对象，其内部都会连续存放三个类型完全不同的数据值，即一个字符指针、一个字符值、一个长整型数值。

在代码的第 10 行，我们通过括号列表初始化的方式，构造了结构 S 的一个对象 s。通过右上方蓝框中的汇编代码，我们可以看到编译器是如何实现对它的初始化的。**本质上，结构只是对其内部所包含各类数据的一个封装，因此从编译产物的角度来看，只需要把它封装的这些数据连续地存放在内存中即可。**事实也正是如此，对结构 S 内部三个数据的初始化过程，均是由指令 mov 完成的，这些数据被初始化在栈内存中。

结构中的数据项被初始化在内存中，这毋庸置疑，但它们真的“连续”吗？

为了验证这个问题，我们在左侧 C 代码的第 12 行，通过 sizeof 运算符将结构 S 的大小打印了出来。按照结构 S 的定义方式和我们对“连续”一词的理解，它在 x86-64 平台上的大小应该为 17 字节。其中，字符指针 8 字节、字符 1 字节，最后的长整型数值 8 字节。但查看右侧黄框内的汇编代码后，你会发现事实并非如此：每一个结构 S 的对象竟然占用了多达 24 字节的内存。那这是为什么呢？

通过整理对象 s 在初始化时使用的汇编代码，我们可以得到其内部各个成员字段在栈内存中的实际布局情况。经过整理后，可以得到下面这张图：



从左至右，这张图代表着栈内存的增长方向（高地址 -> 低地址）。其中，寄存器 rsp 指向栈顶的低地址，而 rbp 寄存器则指向栈帧开始处的高地址。按照汇编代码中的指令，字符指针 p 位于 [rbp-32] 处，并占用 8 个字节；字符 c 位于 [rbp-24] 处，并占用 1 个字节。而长整型变量 x 则位于 [rbp-16] 处，并占用 8 个字节。

可以看到，编译器实际上并没有按照严格连续的方式来“摆放”这三个数据值，其中，[rbp-25] 到 [rbp-16] 中间的 7 个字节并没有存放任何数据。而编译器这样做的一个重要目的，便是为了“数据对齐”。

内存数据对齐

对于现代计算机而言，当内存中需要被读写的数据，其所在地址满足“自然对齐”时，CPU 通常能够以最高的效率进行数据操作。**而所谓自然对齐，是指被操作数据的所在地址为该数据大小的整数倍。**比如在 x86-64 架构中，若一个 `int` 类型的变量，其值在内存中连续存放，且最低有效位字节（LSB）的所在地址为 4 的整数倍，那我们就可以说该变量的值在内存中是对齐的。

自然对齐为什么能够发挥 CPU 最大的内存读取效率呢？这实际上与 CPU 和 MMU（内存管理单元）等内存读写相关核心硬件发展过程中的诸多限制性因素有关。比如，对于某些古老的 Sun SPARC 和 ARM 处理器来说，它们只能访问位于特定地址上的对齐数据，而对于非对齐数据的访问，则会产生异常。相反，有些处理器则能够支持对非对齐数据的访问，但由于设计工艺上的限制，对这些数据的访问需要花费更多的时钟周期。

因此，为了让代码适应不同处理器的“风格”，保证内存中的数据满足自然对齐要求，就成了大多数编译器在生成机器指令时达成的一个默认共识。哪怕在如今的现代 x86-64 处

理器上，访问非对齐数据所产生的性能损耗在大多数情况下已微不足道。

填充字节

让我们再回到之前那个例子。可以看到的是，为了确保对象 s 中所有成员字段在栈内存中都满足自然对齐的要求，编译器会插入额外的“填充字节”，来动态调整结构对象中各个字段对应数据的起始位置。

除此之外，在某些情况下，即使结构对象内各个数据成员都满足自然对齐的要求，额外的填充字节也可能会被添加。比如下面这个例子：

复制代码

```
struct Foo {
  char *p;  // 8 bytes.
  char c;  // 1 bytes.
  // (padding): 7 bytes.
};
```

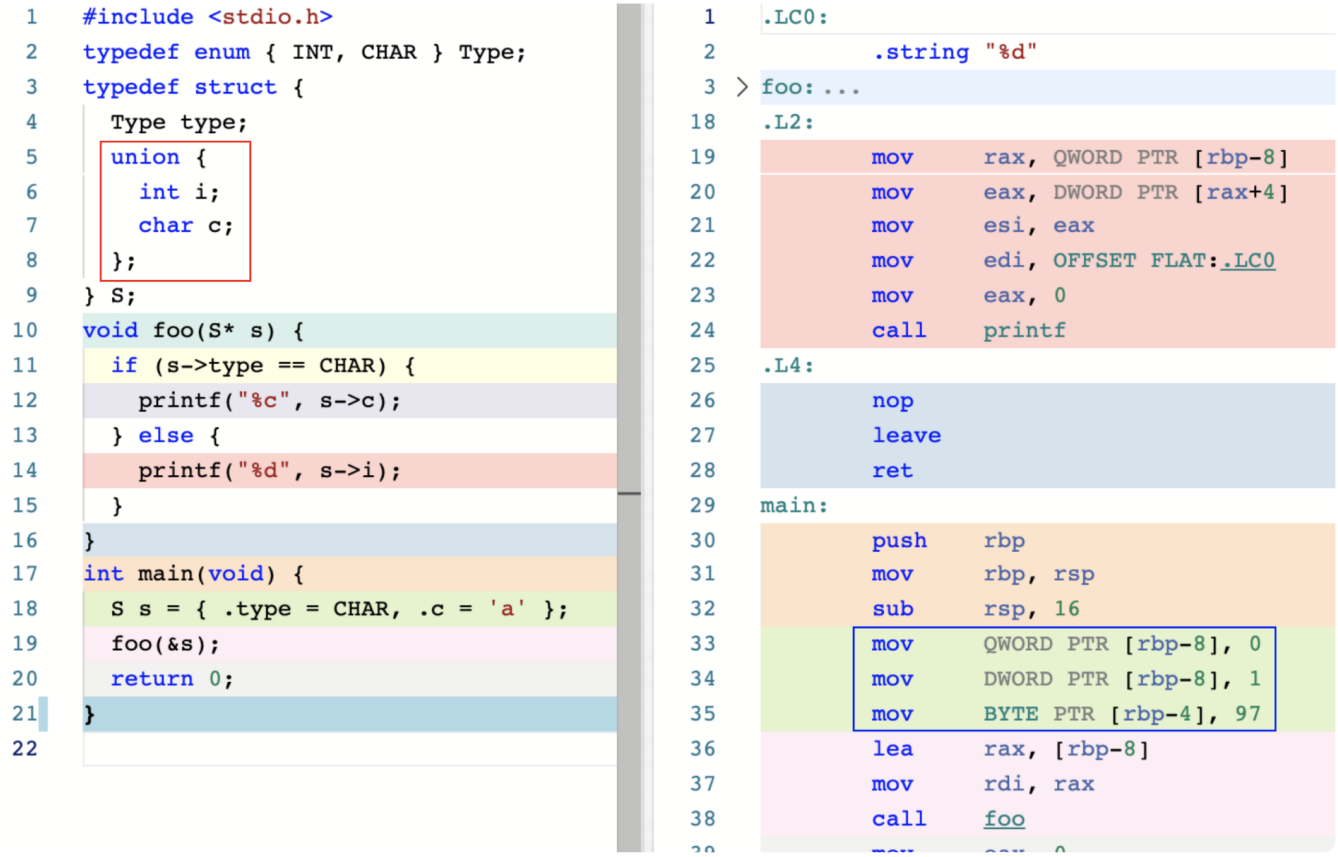
这里可以看到，结构 Foo 中的两个成员字段在默认情况下已经满足自然对齐的要求（假设字符指针 p 的存放起始位置满足 8 字节对齐）。但实际上，在通过 `sizeof` 运算符对它进行求值时，我们会得到 16 字节大小的结果，而非直观的 9 字节。

之所以会出现这样的现象，就是因为编译器想要保证这一点：当结构对象被连续存放时（比如通过数组），前一个对象的结束位置正好可以满足后一个对象作为起始位置时的自然对齐要求。而这也就要求结构对象本身的大小必须是其内部最大成员大小的整数倍。因此，编译器会在结构最后一个成员的后面再填充适当字节，以满足这个条件。可以说，在这种情况下的结构对象，已经满足了在不同场景下的自然对齐条件，因此，此时的结构大小也会被作为 `sizeof` 运算符的最终计算结果。

联合

最后，我们再来看看 C 语言中的第三种功能强大的数据类型，“联合（Union）”。联合与“结构”在语法上的使用方式十分类似，只不过要把对应的语法关键字从 `struct` 更换为 `union`。

除此之外，二者还有一个较大的区别，我们可以从“联合”这个名字谈起。顾名思义，“联合”就意味着定义在该结构内的所有数据字段，将会联合起来共享同一块内存区域。还是先来看一段代码：



```c
#include <stdio.h>
typedef enum { INT, CHAR } Type;
typedef struct {
  Type type;
  union {
    int i;
    char c;
  };
} S;
void foo(S* s) {
  if (s->type == CHAR) {
    printf("%c", s->c);
  } else {
    printf("%d", s->i);
  }
}
int main(void) {
  S s = { .type = CHAR, .c = 'a' };
  foo(&s);
  return 0;
}
```

```asm
.LC0:
        .string "%d"
foo: ...
.L2:
        mov     rax, QWORD PTR [rbp-8]
        mov     eax, DWORD PTR [rax+4]
        mov     esi, eax
        mov     edi, OFFSET FLAT:.LC0
        mov     eax, 0
        call    printf
.L4:
        nop
        leave
        ret
main:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     QWORD PTR [rbp-8], 0
        mov     DWORD PTR [rbp-8], 1
        mov     BYTE PTR [rbp-4], 97
        lea     rax, [rbp-8]
        mov     rdi, rax
        call    foo
```

这里，在左侧的 C 代码中，我们使用 “Tagged Union” 的模式对联合进行了封装。**与结构不同，对于每一个单独的联合对象来说，在某一时刻其内部哪一个字段正在生效，我们无从得知。**因此，Tagged Union 的使用方式要求我们为每一个联合设置单独的“标签”，用来明确指出当前联合内部正在生效的字段。在这种情况下，我们便需要将这个标签与联合进行封装，来将它们进行“绑定”。

可以看到，这里在结构 S 内部，枚举类型字段 type 就是用来标记当前匿名联合内部所存放的数据种类的。而在紧接着的匿名联合内部，整型成员 i 与字符成员 c 则共享该联合的内存空间。这便是 Tagged Union 在 C 语言中的基本使用方式。

一个联合对象的大小同该联合内部定义时所包含最大成员的大小相同，因此在上面这个例子中，结构 S 中的匿名联合大小便与联合定义内整型参数 i 的大小相同。这个大小在 x86-64 平台上为 4 字节。

从图片右侧蓝框内的汇编代码中，我们也可以得到相同的结论。第一行代码将整个结构对象 s 所占用的 8 字节空间全部置零，来为后续的匿名联合对象赋值做准备；第二行代码将枚举类型 CHAR 对应的值 1 赋值给结构对象 s 内的枚举字段 type；第三行代码将字符“a” 对应的值 97 存放到结构对象 s 内的匿名联合对象中。这里可以看到，指令 mov 在进行数据传送时，在目的地参数中使用了 BYTE，也就是“取出”了联合对象所占用的 4 字节空间中的 1 个字节，将其作为存放字符数值的目标内存空间。

总结

好了，讲到这里，今天的内容也就基本结束了，最后我来给你总结一下。

这一讲我主要围绕着 C 语言中的枚举、结构与联合这三种数据类型展开了介绍，和你一起探究了它们在机器指令层面的具体实现方式。

枚举这种数据类型，用于表示可取值范围有限的抽象实体。枚举类型中的枚举值又被称为“具名整型”，因此在 C 代码中，它可以直接被当作整数值来使用。同样地，在编译器生成的代码中，枚举值将被直接替换为对应的整数值。但需要注意的是，我们要在进行 C 编码时保证枚举值和它对应的整数值不被乱用。

结构是一种用于组织异构数据的复合数据类型。在结构中，所有定义的数据字段在内存中按顺序排列。为了保证结构中各个字段最高效的数据访问速度，编译器在内存中布局这些字段数据时，会保证它们的起始地址满足自然对齐的标准。因此，结构中字段的不同定义顺序将直接影响结构对象的实际内存占用大小，而这也是我们进行程序优化的一个重要切入点。

联合是一种特殊的复合数据类型，在其内部定义的所有数据字段将占用同一块内存空间。联合对象的实际大小与其内部所定义最大字段的大小相同。默认情况下，我们无法从外部得知一个联合对象中正在“生效”的字段类型，因此 Tagged Union 的使用方式便成为主流。将用来标识生效字段的枚举类型与联合进行“打包”，我们就可以在使用联合对象前进行相应的判断和准备，而这也为应用程序的健壮性打下了基础。

思考题

最后，我们一起做个思考题。

试着用与这一讲中类似的方式进行分析：下面这个结构体在经过 sizeof 运算符计算后，所得到的大小是多少？

复制代码

```
struct {
  short a;
  char b;
  char c;
  int* d;
  union {
    double e;
    int f;
  };
};
```

今天的课程就结束了，希望可以帮助到你，也希望你在下方的留言区和我一起讨论。同时，欢迎你把这节课分享给你的朋友或同事，我们一起交流。

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 3 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 06 | 代码封装（下）：函数是如何被调用的？

下一篇 08 | 操控资源：指针是如何灵活使用内存的？

精选留言 (5)

写留言

Lijie 民

2021-12-22

24

sky

2021-12-22

问答题，X86_64架构下，24字节。综合考虑两个点：
1.结构体 内部成员 自然对齐；
2.结构体数组的内存对齐。

作者回复: 正确！

liu_liu

2021-12-22

a - 2 字节
b - 1
c - 1
padding - 4
d - 8...
展开

作者回复: 很好的问题！下面是回答：

1. 这里的 ".LC0[rip]" 实际上是汇编代码中的一种通过 rip 寄存器来进行相对寻址的方式，一般也称为 pc-relative 或 rip-relative。rip 寄存器中存放着下一条将要执行的指令。而这里该条指令的寻址结果便是 .LC0 这个 label 的开头。
2. .long 是一个汇编指示标记，用于让汇编器在当前这个位置保留一个 4 字节的值。这里第一个 .long 0 是 1.25 对应二进制表示的低 32 位；第二个 .long 1072955392 则是对应高 32 位的十进制值。

共 2 条评论

pedro

2021-12-22

x86_64 的平台下，sizeof 得到的大小为：24 = 8 + 8 + 8，
加上 packed 属性后，sizeof 为 20 = (1 + 1 + 2) + 8 + 8。

作者回复: 没错！

张贺
2021-12-22

40个字节

作者回复: 不对哦，再看看？